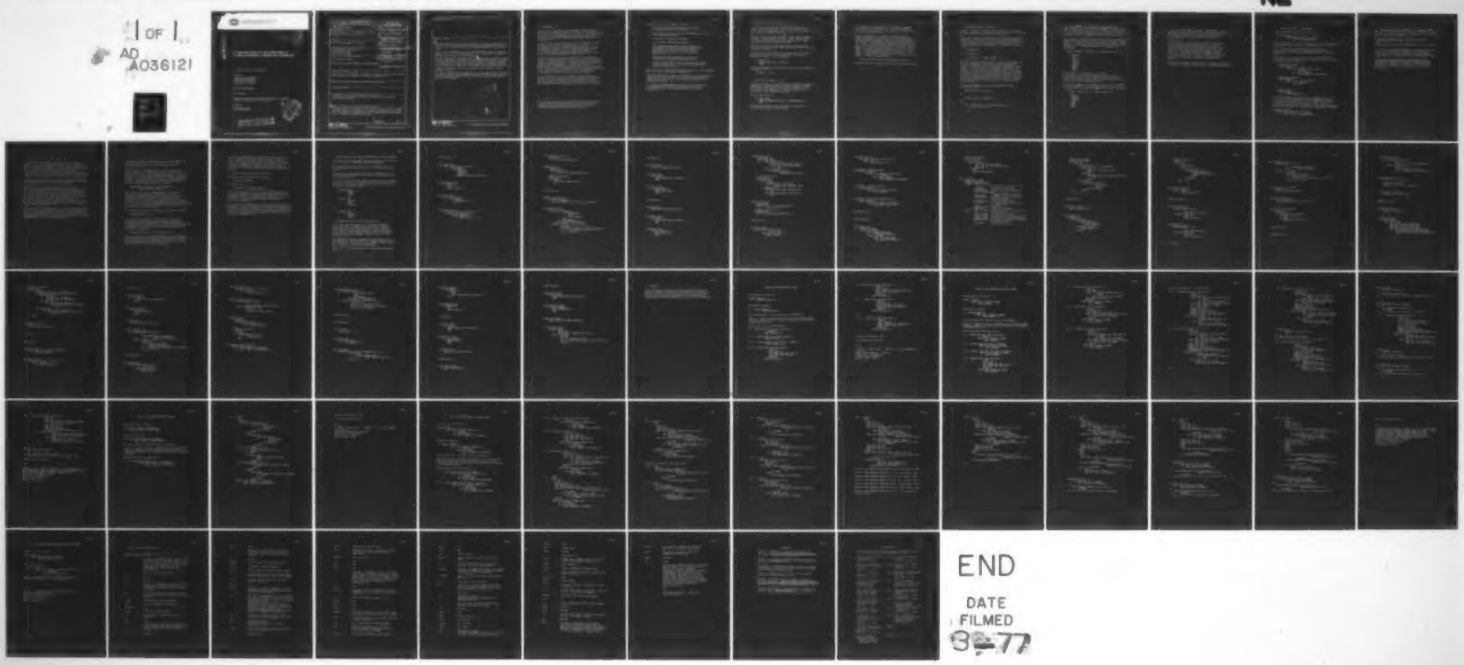
AD-A036 121 STANFORD RESEARCH INST MENLO PARK CALIF F/G 9/2
A COMPUTER PROOF OF THE CORRECTNESS OF A SIMPLE OPTIMIZING COMP--ETC(U)
JAN 77 R S BOYER, J S MOORE N00014-75-C-0816
UNCLASSIFIED SRI-TR-5 NL

1 OF 1
AD
A036121



END
DATE
FILMED
8-77

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

Technical Report 5 January 1977

(10)

A COMPUTER PROOF OF THE CORRECTNESS OF A SIMPLE OPTIMIZING COMPILER FOR EXPRESSIONS

By: ROBERT S. BOYER and J STROTHER MOORE

Prepared for:

OFFICE OF NAVAL RESEARCH
DEPARTMENT OF THE NAVY
ARLINGTON, VIRGINIA 22217
Contract Monitor: Dr. Dick Lau

CONTRACT N00014-75-C-0816

SRI Project 4079

Approved by:

JACK GOLDBERG, *Director*
Computer Science Laboratory

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
N00014-75-C-0816-~~SRI 5~~

2. GOVT ACCESSION NO.

3. ~~RECIPIENT'S CATALOG NUMBER~~
(14) SRI-TR-5

4. TITLE (and ~~Subtitle~~)
(6) A Computer Proof of the Correctness of a Simple Optimizing Compiler for Expressions.

5. TYPE OF REPORT & PERIOD COVERED
(9) Technical ~~Documentary~~ rept.,

6. PERFORMING ORG. REPORT NUMBER
~~N00014-75-C-0816-~~SRI 5

7. AUTHOR(s)
(10) Robert S. Boyer ~~and~~ J Strother Moore

8. CONTRACT OR GRANT NUMBER(s)
(15) N00014-75-C-0816, F44620-73-C-0068

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, CA 94025

10. ~~PROGRAM ELEMENT, PROJECT, TASK~~ AREA & WORK UNIT NUMBERS
~~NRO 49-378~~

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research

12. REPORT DATE
(11) Jan~~uary 19~~77

13. NO. OF PAGES
(12) 62 p.

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)
Distribution of this document is unlimited It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
automatic theorem proving, program verification, structural induction, McCarthy-Painter theorem

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This paper presents an automatic computer proof of the correctness of a simple optimizing compiler for expressions. The proof was produced by a new theorem prover, resembling the Boyer-Moore Pure LISP Theorem Prover but operating on a much richer domain of functions and objects.

DDC

332 500

bpg

DD FORM 1 JAN 73 1473
EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20. ABSTRACT (Continued)

The main theorem proved is that after executing the result of optimizing and compiling a form, the machine's push-down stack is configured just as it would be if one merely pushed the value of the unoptimized form onto the push-down stack.

This result is established by first proving three lemmas: that the optimization phase is correct, that the code generator is correct, and that the optimizer produces legal input for the code generator.

The lemma stating the correctness of the code generator for our push-down stack machine is analogous to the McCarthy-Painter theorem (which was about a machine with addressable memory locations). The system proves the lemma by induction on the structure of the form being compiled, appealing to a single previously proved lemma about the behavior of the "hardware" on a sequence of instructions.

Except for defining the functions and stating the five theorems, no human guidance is required.

This paper briefly describes some aspects of the new theorem prover, informally describes the functions and theorems involved in the compiler correctness proof, and contains machine generated listings of the axioms, function definitions, and the proofs themselves.

1. INTRODUCTION

In 1972, while at the University of Edinburgh, we began work on a theorem prover for primitive recursive functions in the domain of binary trees with the single atom NIL at the tips. This theorem prover became known as the "Boyer-Moore Pure LISP Theorem Prover" and could prove a wide variety of simple theorems about functions in the above domain [1].

In the four years since we began work on that theorem prover we have catalogued a large number of inadequacies in its theory, heuristics, philosophy, and implementation (as well as an equally large number of good ideas in each of those departments). During the past three months we have finally begun the construction of a new theorem prover designed to remedy the inadequacies of the original while taking advantage of the good ideas.

From the outside, the most radical change is that the theory with which the theorem prover deals is that of total (but not necessarily primitive) recursive functions in a domain of axiomatically specified finitely constructable objects. The theorem prover contains built-in knowledge about IF-THEN-ELSE (as the only logical connective), EQUAL (standard equality), typed n-tuples (useful in the introduction of new data types), and recursion/induction. All aspects of the theorem prover, from its symbolic evaluator to its induction mechanism, are completely controlled by axioms and/or proved lemmas.

The new theorem prover is like the old one in being completely automatic (subject to the above caveat concerning axioms and lemmas). It also shares the philosophy that one should simplify a formula whenever possible and be able to invoke induction as a last resort. The cross-fertilization and generalization heuristics have been kept.

The research described here has been supported by the Office of Naval Research under Contract N00014-75-C-0816, the National Science Foundation under Grant DCR72-03737A01, and the Air Force Office of Scientific Research under Contract F44620-73-C-0068.

Many new ideas have been introduced, including:

- A mechanism for introducing new data types (however, the language itself is still untyped).
- A very fast and powerful facility for determining the type of an expression or function.
- A mechanism for introducing new axioms.
- A very powerful lemma-driven simplifier.
- A new induction mechanism which uses the principle of lexicographic ordering and axioms and/or lemmas about well-founded partial orders to produce structural inductions (in the general sense in which Burstall defined it [2]).
- A mechanism for eliminating many irrelevant details from automatically generated sub-goals (lemmas).
- A "proof monitor" which allows the use of previously proved sub-goals during a proof, and which can inform the theorem prover that it is failing.

Since we are still constructing the new theorem prover and daily changing many basic ideas, we have not yet documented it.

The new theorem prover is capable of proving all the old theorems (when reformulated in the new theory) and is currently being tested on new theorems (indeed, new theories, since the new version admits the introduction of axioms).

The first new theorem the system was tried on was the correctness of a simple optimizing compiler for expressions. We believe its automatic proof of this result augurs well for its future capabilities.

2. AN INFORMAL DISCUSSION OF THE THEORY

Section 7 of this document is a machine-generated listing of the axioms, definitions, and lemmas involved in the proof of the correctness of our optimizing compiler. This Section is devoted to an informal discussion of the theory.

The theory contains six types of objects: TRUE, FALSE, non-negative integers (constructed from 0 with ADD1), literal atoms ("words") (constructed with PACK), ordered pairs (constructed with CONS), and push-down stacks (constructed with PUSH).

The axioms in Section 7 define these objects formally, specify the equality relations between them, and define the various functions for manipulating them.

It should be noted that induction on objects such as the ordered pairs is permitted by virtue of axioms such as the one named CDR-LESSP in Section 7. That axiom states:

```
(IF (LISTP X)
    (LESSP (COUNT (CDR X)) (COUNT X))
    TRUE).
```

Since LESSP is known to be a well-founded ordering this axiom permits an induction such as:

```
~(LISTP X) -> (P X)

        &

(LISTP X) & (P (CDR X)) -> (P X)
```

to prove (P X) for all X. The theorem prover's induction mechanism can chain many such axioms together to derive, for example, that when (LISTP X) and (LISTP (CDR X)) then (COUNT (CDR (CDR X))) is less than (COUNT X), thus justifying an inductive hypothesis about (CDR (CDR X)) (under appropriate conditions).

Proving a lemma such as:

```
(IF (LESSP I MAX)
    (LESSP (DIFFERENCE MAX (ADD1 I)) (DIFFERENCE MAX I))
    TRUE)
```

informs the theorem prover that it is sound to "induct up" by (possibly nested) ADD1's to some maximum.

The theorem prover also knows that it can choose to instantiate free variables in the hypothesis of an inductive assumption, and, indeed, can choose several such instantiations. This fact is used in the compiler proof.

We have implemented certain cosmetic functions allowing the convenient expression of constants in the theory. For example, INTERLISP integers appearing in input to the theorem prover are translated into the appropriate ADD1 nests, which are printed back out as digits. The word NIL is translated into (PACK 0) which, in turn, is printed as NIL. Similarly, any word preceded by "'" (or embedded in a QUOTE form) in the input is translated into (PACK n), where n is the number of such distinct words seen so far. Thus, 'PUSHI is translated into (PACK 1), which, in turn, is printed as 'PUSHI. Finally, INTERLISP S-expressions preceded by "'" (or embedded in QUOTE forms) are translated into the appropriate CONS terms, e.g., '(PLUS 1 X) might become:

```
(CONS (PACK 4) (CONS (ADD1 0) (CONS (PACK 5) (PACK 0)))),
```

(depending only upon how many other quoted words had been introduced).

3. THE FUNCTIONS INVOLVED IN THE PROOF

Section 7 contains the formal definitions of all of the functions involved in the proof of correctness of the compiler. In this Section we will briefly describe each of the main functions.

The recursive predicate FORMP recognizes S-expressions representing expressions. For simplicity we only consider compiling numerically valued expressions involving numbers, variables, and the application of binary functions to (the values of) two sub-expressions. We represent expressions as S-expressions in the obvious way. A typical expression is:

```
(3+4)*(X*(Y*6))
```

and is represented as:

```
'(TIMES (PLUS 3 4) (TIMES X (TIMES Y 6))).
```

The function EVAL defines the value of a form in an environment which specifies the values of the variables. We used two undefined functions, GETVALUE and APPLY, to define EVAL. GETVALUE takes two arguments, a variable and an environment, and is supposed to return the value of the variable in that environment. APPLY takes three arguments, a function name and two argument values, and is supposed to return the value of the function on the two arguments. Since both EVAL and the "hardware" use these primitives, their definition was not necessary: regardless of what meaning one attaches to the terms "value of X in ENV" and "the value of fn on x and y," the compiler must produce code which causes the hardware to produce the same result EVAL would.

As an example of EVAL, consider defining GETVALUE as ASSOC and adding the axioms:

```
(APPLY 'PLUS X Y) = (PLUS X Y),
```

and

```
(APPLY 'TIMES X Y) = (TIMES X Y),
```

then:

```
(EVAL '(TIMES (PLUS 3 4) (TIMES X (TIMES Y 6)))
      '((X . 2)(Y . 5)))

= 420.
```

The function OPTIMIZE takes a form and returns another form which will always have the same value (as computed by EVAL) as the input form. OPTIMIZE works by optimizing the arguments to a function call, and then, provided both arguments have been optimized to integers, it replaces the function call by the value of the function on the two integers. That is, OPTIMIZE just performs "constant folding."

The code generator, CODEGEN, takes as input a form to be compiled and an arbitrary list of initial instructions. The second argument is used merely as an accumulator during the compilation. As output, CODEGEN produces a list of "machine instructions" which should be executed in REVERSE order on a "hardware" machine with a push-down stack. For example,

```
   (REVERSE (CODEGEN '(TIMES (PLUS 3 4) (TIMES X (TIMES Y 6))) NIL))

     = '((PUSHI 3)
         (PUSHI 4)
         PLUS
         (PUSHV X)
         (PUSHV Y)
         (PUSHI 6)
         TIMES
         TIMES
         TIMES).
```

The meanings of the instructions are: (PUSHI n) means "push n," (PUSHV x) means "push the value of x in the current environment," and anything else is taken as a function name and means "pop two things off the stack and push the value of the function on the two things popped off."

The compiler, COMPILE, first optimizes its input form with OPTIMIZE, then calls CODEGEN with the optimized form and the initial list of instructions NIL, and returns the REVERSE of the output. For example,

```
     (COMPILE '(TIMES (PLUS 3 4) (TIMES X (TIMES Y 6))))

     = ((PUSHI 7)
        (PUSHV X)
        (PUSHV Y)
        (PUSHI 6)
        TIMES
        TIMES
        TIMES).
```

Finally, the function EXEC is defined as the "hardware." This function takes a list of instructions, a push-down stack, and an environment (specifying variable values) and interprets the instructions in the obvious way. When it has exhausted the list of instructions it returns the final value of the push-down stack.

All function definitions introduced are translated according to the cosmetic conventions mentioned above. In addition, a macro facility is available to permit the input syntax to vary radically from the usual LISP-like notation we prefer. This facility is used to make the definition of EXEC, for example, look like it is written in a dialect of assembly language. Definitions involving PROG, SETQ, GO, and RETURN are translated into recursive definitions. In Section 7 we present both the "pretty" and the translated (recursive) definitions of all functions.

The reader is advised to turn to Section 7 and study the definitions of FORMP, EVAL, OPTIMIZE, COGEGEN, COMPILE, and EXEC before proceeding.

4. AN INFORMAL DISCUSSION OF THE THEOREMS PROVED

The main theorem proved is:

```
(IMPLIES (FORMP X)
         (EQUAL (EXEC (COMPILE X) PDS ENVRN)
                (PUSH (EVAL X ENVRN) PDS))).
```

This says that, when X is a form, executing the output of the (optimizing) compiler merely pushes the value of (unoptimized) X onto the stack.

Each theorem the theorem prover proves is given a name, so that it can be referenced as a lemma in later proofs. We have assigned the name "CORRECTNESS.OF.OPTIMIZING.COMPILER" to the main result above.

This theorem is proved by the invocation of three lemmas which were also proved by the theorem prover (but, of course, posed by the authors). These lemmas and their names are:

```
CORRECTNESS.OF.CODEGEN:
(IMPLIES (FORMP X)
         (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                      PDS ENVRN)
                (PUSH (EVAL X ENVRN)
                      (EXEC (REVERSE INS) PDS ENVRN)))),
```

```
CORRECTNESS.OF.OPTIMIZE:
(IMPLIES (FORMP X)
         (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                (EVAL X ENVRN))),
```

and

```
FORMP.OPTIMIZE:
(IMPLIES (FORMP X)
         (FORMP (OPTIMIZE X))).
```

Note that CORRECTNESS.OF.CODEGEN states that CODEGEN is correct in the sense that executing the reverse of (CODEGEN X INS) produces a stack which has the value of X on the top and the stack produced by executing the reverse of INS underneath. This is just the McCarthy-Painter theorem (for our CODEGEN and EXEC). The theorem prover requires the following lemma in order to prove CORRECTNESS.OF.CODEGEN:

```
SEQUENTIAL.EXECUTION:
     (EQUAL (EXEC (APPEND X Y) PDS ENVRN)
            (EXEC Y (EXEC X PDS ENVRN) ENVRN)).
```

This lemma just says that if one appends two sequences of instructions and executes the result, the push-down stack is the same obtained by executing the second sequence on the stack obtained by executing the first. Having proved this lemma, our theorem prover can automatically prove the correctness of CODEGEN.

CORRECTNESS.OF.OPTIMIZE states that the output of OPTIMIZE always has the same value (under EVAL) as its input.

FORMP.OPTIMIZE merely states that OPTIMIZE produces a form if given one.

It is clear how CORRECTNESS.OF.CODEGEN and CORRECTNESS.OF.OPTIMIZE contribute to the proof of CORRECTNESS.OF.OPTIMIZING.COMPILER. Less obvious is the necessity of backwards chaining through FORMP.OPTIMIZE to relieve the hypothesis of CORRECTNESS.OF.CODEGEN.

The theorem prover proved these five theorems in the order FORMP.OPTIMIZE, CORRECTNESS.OF.OPTIMIZE, SEQUENTIAL.EXECUTION, CORRECTNESS.OF.CODEGEN, and CORRECTNESS.OF.OPTIMIZING.COMPILER. Except for the definitions of the functions concerned and the statement of the theorems (in the order given), no user help was required.

5. A NOTE ON THE McCARTHY-PAINTER COMPILER PROOF

The well-known McCarthy-Painter compiler correctness theorem [5] is analogous to our lemma CORRECTNESS.OF.CODEGEN. Their "hardware" differs from ours in that temporary results are stored in addressable registers rather than a push-down stack. This somewhat complicates the compiler, since it must know in what address it left which result. However, while our compiler only lays down pushes and pops, it should be noted that the proof requires arguing that every pop caused by compiled code is matched by a previous push (of the correct result).

As explained above, while the proof of correctness for the optimizing compiler requires four lemmas, the theorem prover only needs one lemma to establish automatically the correctness of CODEGEN.

The McCarthy-Painter proof was first done by hand in 1967. We know of three mechanically checked proofs of the theorem. The first was apparently by Diffie [4] with his proof checker for first order predicate calculus. The second was by Milner and Weyhrauch [6] with their LCF proof checker. The third was by Cartwright [3] with his interactive verifier for TYPED LISP. Because of the proof checker orientation of these systems these three proofs required a fair amount of user guidance. We are aware that Milner (now at the Computer Science Department of the University of Edinburgh) is working on a more automatic LCF theorem prover and we believe his system might now be capable of producing this proof.

It should also be noted that we first proved the correctness of CODEGEN using a function called LITEXEC, which is similar to the LIT function employed in Burstall's proof [2]. We then had the theorem prover prove the equivalence of EXEC and LITEXEC. Later we discovered that, with the SEQUENTIAL.EXECUTION lemma, the theorem prover could prove the correctness of CODEGEN directly, stated as it is here presented.

6. BUGS DISCOVERED BY ATTEMPTED PROOFS

To our great surprise the first three times we tried to prove these theorems the theorem prover failed in such a way as to make obvious three honest mistakes in our implementation of the functions concerned.

The first bug was in OPTIMIZE. When it finds a sub-expression, such as '(PLUS 3 4) in '(TIMES (PLUS 3 4) X), it replaces the sub-expression by the value of applying the function to the indicated constants. Under the interpretation of APPLY we had in mind, this would yield, in the above case, '(TIMES 7 X). However, originally we did not explicitly assume that our expressions were numerically valued. Consequently, upon trying to prove the correctness of OPTIMIZE, the theorem prover failed on the sub-goal:

```
(IMPLIES (AND (NOT (LISTP Z)) (NOT (NUMBERP Z)))
         (EQUAL (GETVALUE Z ENVRN) Z)).
```

This goal arose (after generalizing (APPLY FN I J) to Z) from considering the possibility that OPTIMIZE encountered an optimizable sub-expression, such as (FN I J) where I and J were numbers, and such that (APPLY FN I J) produced a non-number, non-list answer. For example, if the application of FN to 3 and 4 produces the literal atom Y, then optimizing (G (FN 3 4) X) produces (G Y X). Thus, evaluating the optimized form would apply G to the values of Y and X, while evaluating the unoptimized form would apply G to the literal atom Y and the value of X.

We remedied the situation by introducing the only axiom about APPLY:

```
(NUMBERP (APPLY FN X Y)).
```

The second bug was in EXEC's handling of function calls. When it was time to pop the stack and apply a function, EXEC popped the stack into AC1, then popped the next thing into AC2, and then pushed (APPLY FN AC1 AC2). However, upon trying to prove the correctness of CODEGEN, the theorem prover, after a generalization, stopped because it could not prove:

```
(EQUAL (APPLY FN AC1 AC2) (APPLY FN AC2 AC1)).
```

That is, it had to prove that all the functions in our expressions were commutative! The first definition of EXEC swapped the arguments to functions because we had overlooked the fact that the value of the first argument expression is pushed first, and hence is the second thing on the stack at the time of a function call.

The third bug we discovered was in FORMP. Originally we believed that it did not matter what the CAR of a form was. This was because APPLY, being undefined, could be understood to make some sense out of any possible "function name." Therefore, the original definition of FORMP did not have the (NLISTP (CAR X)) check. This permitted forms with non-atomic function names. Upon trying to prove the correctness of CODEGEN the theorem prover stopped on the subgoal:

```
(EQUAL X (APPLY (LIST 'PUSHI X) ARG1 ARG2)).
```

This goal arises by considering a form such as:

```
'((PUSHI X) ARG1 ARG2).
```

The compiled code is:

```
'((PUSHV ARG1) (PUSHV ARG2) (PUSHI X)),
```

(since function names are laid down as single instructions which supposedly cause EXEC to pop the stack twice and push the result of applying the function). However, if the CAR of the form can be confused with a PUSHI or PUSHV instruction, EXEC would not call APPLY (as EVAL would) but merely PUSH the indicated value. Hence the above sub-goal.

It is interesting to note that all three of these bugs escaped our rudimentary testing of the functions. In particular, we had confirmed, by example, that all the functions were "correct," by testing them on a variety of arithmetic expressions. Unfortunately, all of our test cases were with numerically valued commutative functions with atomic names, such as TIMES and PLUS.

7. THE THEORY BEHIND THE PROOF OF "CORRECTNESS.OF.OPTIMIZING.COMPILER"

Below we give the axioms, lemmas, function definitions and abbreviations used in the proof of CORRECTNESS.OF.OPTIMIZING.COMPILER.

The expression (IF x y z) means "if x is not (FALSE), then y, else z." The expression (EQUAL x y) means "if x is y then (TRUE), else (FALSE)."

The axioms about TRUE, FALSE, 0, ADD1, SUB1, and NUMBERP were all typed into the system by hand and are part of the standard "loadup" for the theorem prover.

The axioms about PACK, CONS and PUSH (and their accessors, recognizers, and default values) were all generated automatically by a mechanism allowing the introduction of typed n-tuples. In particular, these axioms were generated by incanting:

```
     (ADD.SHELL PACK
                (UNPACK)
                LITATOM
                (NIL)),

     (ADD.SHELL CONS
                (CAR CDR)
                LISTP
                (NIL NIL)),
```

and

```
     (ADD.SHELL PUSH
                (TOP POP)
                STACKP
                (1 NIL))
```

to the INTERLISP system running the theorem prover.

All type-in to the theorem prover is translated into functional form. QUOTE, SETQ, GO, RETURN, COND, LIST, PROG, and PROGN are handled specially. Any form beginning with a macro is replaced by the result of evaluating the body of the macro's definition in an environment in which the macro argument is bound to the CDR of the form and the variable PROGBODY is bound to the list of forms following the form.

When printing out formulas we employ certain abbreviations. In the following read (TRUE) for TRUE and T, (FALSE) for FALSE, (ADD1 n) for n+1 when n is a non-negative integer, (PACK 0) for NIL, (PACK 1) for (QUOTE PUSHI) and (PACK 2) for (QUOTE PUSHV).

Section 9 is a cross reference table for the axioms and definitions below.

```
ADD1 (Primitive)


ADD1.EQUAL (Axiom)
     (EQUAL (EQUAL (ADD1 X) (ADD1 Y))
            (IF (NUMBERP X)
                (IF (NUMBERP Y)
                    (EQUAL X Y)
                    (EQUAL X 0))
                (IF (NUMBERP Y) (EQUAL Y 0) TRUE)))


ADD1.NNUMBERP (Axiom)
     (IF (NUMBERP X)
         TRUE
         (EQUAL (ADD1 X) 1))


ADD1.SUB1 (Axiom)
     (EQUAL (ADD1 (SUB1 X))
            (IF (NUMBERP X)
                (IF (EQUAL X 0) 1 X)
                1))


ADD1.TYPE.NO (Axiom)
     (AND (EQUAL (TYPE.NO (ADD1 SUB1)) 2)
          (AND (EQUAL (TYPE.NO 0) 2)
               (EQUAL (NUMBERP X)
                      (EQUAL (TYPE.NO X) 2))))
```

```
AND (Translated definition)
     (LAMBDA (P Q)
      (IF P (IF Q TRUE FALSE) FALSE))



APPEND (Definition)
     (LAMBDA (X Y)
      (COND ((NLISTP X) Y)
            (T (CONS (CAR X) (APPEND (CDR X) Y)))))

     (Translated definition)
     (LAMBDA (X Y)
      (IF (LISTP X)
          (CONS (CAR X) (APPEND (CDR X) Y))
          Y))



APPLY (Primitive)



ASSEMBLE (INTERLISP Macro)
     (X (CONS (QUOTE PROG) X)
        (Comment: ASSEMBLE is just another name for PROG.))



CAIE (INTERLISP Macro)
     (X (PROG1 (LIST (QUOTE IF)
                     (LIST (QUOTE EQUAL)
                           (CAR X)
                           (KWOTE (CADR X)))
                     FALSE
                     (CAR PROGBODY))
               (SETQ PROGBODY (CDR PROGBODY)))
         (Comment: (CAIE ac val) instr1 instr2 ...
                   turns into
                   (IF (EQUAL ac (QUOTE val)) FALSE instr1) instr2 ...
                   i.e., it skips the next instruction if ac is
                   equal to (QUOTE val)))
```

```
CAR (Primitive)


CAR.CONS (Axiom)
     (EQUAL (CAR (CONS CAR CDR)) CAR)


CAR.LESSP (Axiom)
     (IF (LISTP X)
         (LESSP (COUNT (CAR X)) (COUNT X))
         TRUE)


CAR.NLISTP (Axiom)
     (IF (NOT (LISTP X))
         (EQUAL (CAR X) NIL)
         TRUE)


CDR (Primitive)


CDR.CONS (Axiom)
     (EQUAL (CDR (CONS CAR CDR)) CDR)


CDR.LESSP (Axiom)
     (IF (LISTP X)
         (LESSP (COUNT (CDR X)) (COUNT X))
         TRUE)


CDR.NLISTP (Axiom)
     (IF (NOT (LISTP X))
         (EQUAL (CDR X) NIL)
         TRUE)
```

```
CODEGEN (Definition)
     (LAMBDA (FORM INS)
      (COND ((NLISTP FORM)
             (COND ((NUMBERP FORM)
                    (CONS (LIST (QUOTE PUSHI) FORM) INS))
                   (T (CONS (LIST (QUOTE PUSHV) FORM)
                            INS))))
            (T (CONS (CAR FORM)
                     (CODEGEN (CADDR FORM)
                              (CODEGEN (CADR FORM) INS)))))))

     (Translated definition)
     (LAMBDA (FORM INS)
      (IF (LISTP FORM)
          (CONS (CAR FORM)
                (CODEGEN (CAR (CDR (CDR FORM)))
                         (CODEGEN (CAR (CDR FORM)) INS)))
          (IF (NUMBERP FORM)
              (CONS (CONS (QUOTE PUSHI) (CONS FORM NIL))
                    INS)
              (CONS (CONS (QUOTE PUSHV) (CONS FORM NIL))
                    INS))))


COMPILE (Definition)
     (LAMBDA (FORM)
      (REVERSE (CODEGEN (OPTIMIZE FORM) NIL)))

     (Translated definition)
     (LAMBDA (FORM)
      (REVERSE (CODEGEN (OPTIMIZE FORM) NIL)))


CONS (Primitive)


CONS.EQUAL (Axiom)
     (EQUAL (EQUAL (CONS CAR CDR)
                   (CONS CAR' CDR'))
            (AND (EQUAL CAR CAR')
                 (EQUAL CDR CDR')))
```

```
CONS.TYPE.NO (Axiom)
     (AND (EQUAL (TYPE.NO (CONS CAR CDR)) 4)
          (EQUAL (LISTP X)
                 (EQUAL (TYPE.NO X) 4)))


CORRECTNESS.OF.CODEGEN (Theorem)
     (IMPLIES (FORMP X)
              (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                           PDS ENVRN)
                     (PUSH (EVAL X ENVRN)
                           (EXEC (REVERSE INS) PDS ENVRN))))


CORRECTNESS.OF.OPTIMIZE (Theorem)
     (IMPLIES (FORMP X)
              (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                     (EVAL X ENVRN)))


CORRECTNESS.OF.OPTIMIZING.COMPILER (Theorem)
     (IMPLIES (FORMP X)
              (EQUAL (EXEC (COMPILE X) PDS ENVRN)
                     (PUSH (EVAL X ENVRN) PDS)))


COUNT (Primitive)


EQUAL (Primitive)


EVAL (Definition)
     (LAMBDA (FORM ENVRN)
      (COND ((NLISTP FORM)
             (COND ((NUMBERP FORM) FORM)
                   (T (GETVALUE FORM ENVRN))))
            (T (APPLY (CAR FORM)
                      (EVAL (CADR FORM) ENVRN)
                      (EVAL (CADDR FORM) ENVRN)))))
```

```
    (Translated definition)
    (LAMBDA (FORM ENVRN)
     (IF (LISTP FORM)
         (APPLY (CAR FORM)
                (EVAL (CAR (CDR FORM)) ENVRN)
                (EVAL (CAR (CDR (CDR FORM))) ENVRN))
         (IF (NUMBERP FORM)
             FORM
             (GETVALUE FORM ENVRN))))


EXEC (Definition)
    (LAMBDA (PC PDS ENVRN)
     (ASSEMBLE (INSTR AC1 AC2)
      LOOP     (SKIPTYPE PC LISTP)
                                   (* skip next instr if PC is a list)
               (RETURN PDS)        (* exit with current PDS)
               (ILDI INSTR PC)     (* fetch next instr and bump pc)
               (SKIPTYPE INSTR NLISTP)
                                   (* skip if INSTR is not a list)
               (JUMP DECODE)       (* INSTR is a list, go decode it)
               (POPARG AC2 PDS)    (* INSTR represents a fn call. Pop
                                   second arg into AC2.)
               (POPARG AC1 PDS)    (* Pop first arg into AC1)
               (XCT INSTR)         (* execute INSTR on AC1 and AC2 and
                                   put result in AC1)
               (PUSHARG AC1 PDS)   (* push AC1)
               (JUMP LOOP)         (* go fetch next instr)
      DECODE   (HLRZ AC1 INSTR)    (* INSTR is a LISTP type instr. Fetch
                                   opcode into AC1)
               (HRRZ AC2 INSTR)    (* fetch operand into AC2)
               (CAIE AC1 PUSHI)    (* Skip next instr if opcode is PUSHI)
               (LOAD@ AC2 AC2)     (* Clobber AC2 with the value of AC2)
               (PUSHARG AC2 PDS)   (* Push AC2 -- will be either operand
                                   or value of operand)
               (JUMP LOOP)         (* go fetch next instr)))
```

```
(Translated definition)
(LAMBDA (PC PDS ENVRN)
 (IF (LISTP PC)
     (IF (LISTP (CAR PC))
         (IF (EQUAL (CAR (CAR PC)) (QUOTE PUSHI))
             (EXEC (CDR PC)
                   (PUSH (CAR (CDR (CAR PC))) PDS)
                   ENVRN)
             (EXEC (CDR PC)
                   (PUSH (GETVALUE (CAR (CDR (CAR PC))) ENVRN)
                         PDS)
                   ENVRN))
         (EXEC (CDR PC)
               (PUSH (APPLY (CAR PC)
                            (TOP (POP PDS))
                            (TOP PDS))
                     (POP (POP PDS)))
               ENVRN))
     PDS))
```

```
FALSE (Primitive)
```

```
FALSE.TYPE.NO (Axiom)
     (EQUAL (TYPE.NO FALSE) 0)
```

```
FORMP (Definition)
     (LAMBDA (X)
      (COND ((NLISTP X) T)
            ((AND (NLISTP (CAR X))
                  (AND (LISTP (CDR X))
                       (LISTP (CDDR X))))
             (AND (FORMP (CADR X))
                  (FORMP (CADDR X))))
            (T FALSE)))
```

```
     (Translated definition)
     (LAMBDA (X)
      (IF (LISTP X)
          (IF (LISTP (CAR X))
              FALSE
              (IF (LISTP (CDR X))
                  (IF (LISTP (CDR (CDR X)))
                      (IF (FORMP (CAR (CDR X)))
                          (FORMP (CAR (CDR (CDR X))))
                          FALSE)
                      FALSE)
                  FALSE))
          TRUE))
```

```
FORMP.OPTIMIZE (Theorem)
     (IMPLIES (FORMP X)
              (FORMP (OPTIMIZE X)))
```

```
GETVALUE (Primitive)
```

```
HLRZ (INTERLISP Macro)
     (X (SUBPAIR (QUOTE (AC X))
                 X
                 (QUOTE (SETQ AC (CAR X))))
        (Comment: (HLRZ AC X)
                  turns into
                  (SETQ AC (CAR X))))
```

```
HRRZ (INTERLISP Macro)
     (X (SUBPAIR (QUOTE (AC X))
                 X
                 (QUOTE (SETQ AC (CADR X))))
        (Comment: (HRRZ AC X)
                  turns into
                  (SETQ AC (CADR X))))
```

```
IF (Primitive)
```

```
ILDI (INTERLISP Macro)
     (X (SUBPAIR (QUOTE (INS PC))
                 X
                 (QUOTE (PROGN (SETQ INS (CAR PC))
                               (SETQ PC (CDR PC)))))
        (Comment: (ILDI INS PC)
                  turns into
                  (PROGN (SETQ INS (CAR PC))
                         (SETQ PC (CDR PC)))
                  i.e., it fetches the next instruction into INSTR and
                  bumps the PC by one.))
```

```
IMPLIES (Translated definition)
     (LAMBDA (P Q)
      (IF P (IF Q TRUE FALSE) TRUE))
```

```
JUMP (INTERLISP Macro)
     (X (CONS (QUOTE GO) X)
        (Comment: JUMP is another name for GO.))
```

```
LESSP (Translated definition)
     (LAMBDA (X Y)
      (IF (EQUAL X 0)
          (IF (EQUAL Y 0) FALSE TRUE)
          (IF (EQUAL Y 0)
              FALSE
              (LESSP (SUB1 X) (SUB1 Y)))))
```

LISTP (Primitive)

LITATOM (Primitive)

```
LOAD@ (INTERLISP Macro)
     (X (SUBPAIR (QUOTE (AC VAR))
                 X
                 (QUOTE (SETQ AC (GETVALUE VAR ENVRN))))
        (Comment: (LOAD@ ac var)
                  turns into
                  (SETQ ac (GETVALUE var ENVRN))
                  i.e., it loads ac with the value of var.))


NLISTP (Definition)
     (LAMBDA (X) (NOT (LISTP X)))

     (Translated definition)
     (LAMBDA (X) (IF (LISTP X) FALSE TRUE))


NOT (Translated definition)
     (LAMBDA (P) (IF P FALSE TRUE))


NUMBERP (Primitive)


NUMBERP.APPLY (Axiom)
     (NUMBERP (APPLY FN X Y))


OPTIMIZE (Definition)
     (LAMBDA (FORM)
      (PROG (TEMP1 TEMP2)
            (COND ((NLISTP FORM) (RETURN FORM)))
            (SETQ TEMP1 (OPTIMIZE (CADR FORM)))
            (SETQ TEMP2 (OPTIMIZE (CADDR FORM)))
            (COND ((AND (NUMBERP TEMP1) (NUMBERP TEMP2))
                   (RETURN (APPLY (CAR FORM) TEMP1 TEMP2)))
                  (T (RETURN (LIST (CAR FORM) TEMP1 TEMP2))))))
```

```
     (Translated definition)
     (LAMBDA (FORM)
      (IF (LISTP FORM)
          (IF (NUMBERP (OPTIMIZE (CAR (CDR FORM))))
              (IF (NUMBERP (OPTIMIZE (CAR (CDR (CDR FORM)))))
                  (APPLY (CAR FORM)
                         (OPTIMIZE (CAR (CDR FORM)))
                         (OPTIMIZE (CAR (CDR (CDR FORM)))))
                  (CONS (CAR FORM)
                        (CONS (OPTIMIZE (CAR (CDR FORM)))
                              (CONS (OPTIMIZE (CAR (CDR (CDR FORM))))
                                    NIL))))
              (CONS (CAR FORM)
                    (CONS (OPTIMIZE (CAR (CDR FORM)))
                          (CONS (OPTIMIZE (CAR (CDR (CDR FORM))))
                                NIL))))
          FORM))
```

```
OR (Translated definition)
     (LAMBDA (P Q)
      (IF P TRUE (IF Q TRUE FALSE)))
```

```
P (Primitive)
```

```
PACK (Primitive)
```

```
PACK.EQUAL (Axiom)
     (EQUAL (EQUAL (PACK UNPACK) (PACK UNPACK'))
            (EQUAL UNPACK UNPACK'))
```

```
PACK.TYPE.NO (Axiom)
     (AND (EQUAL (TYPE.NO (PACK UNPACK)) 3)
          (EQUAL (LITATOM X)
                 (EQUAL (TYPE.NO X) 3)))
```

```
POP (Primitive)


POP.LESSP (Axiom)
     (IF (STACKP X)
         (LESSP (COUNT (POP X)) (COUNT X))
         TRUE)


POP.NSTACKP (Axiom)
     (IF (NOT (STACKP X))
         (EQUAL (POP X) NIL)
         TRUE)


POP.PUSH (Axiom)
     (EQUAL (POP (PUSH TOP POP)) POP)


POPARG (INTERLISP Macro)
     (X (SUBPAIR (QUOTE (AC STACK))
                 X
                 (QUOTE (PROGN (SETQ AC (TOP STACK))
                               (SETQ STACK (POP STACK))))))
        (Comment: (POPARG AC STACK)
                  turns into
                  (PROGN (SETQ AC (TOP STACK))
                         (SETQ STACK (POP STACK)))
                  i.e., it pops the top of STACK into AC decrementing
                  the stack pointer.))


PUSH (Primitive)


PUSH.EQUAL (Axiom)
     (EQUAL (EQUAL (PUSH TOP POP)
                   (PUSH TOP' POP'))
            (AND (EQUAL TOP TOP')
                 (EQUAL POP POP')))
```

```
PUSH.TYPE.NO (Axiom)
      (AND (EQUAL (TYPE.NO (PUSH TOP POP)) 5)
           (EQUAL (STACKP X)
                  (EQUAL (TYPE.NO X) 5)))


PUSHARG (INTERLISP Macro)
      (X (SUBPAIR (QUOTE (AC STACK))
                  X
                  (QUOTE (SETQ STACK (PUSH AC STACK))))
         (Comment: (PUSHARG AC STACK)
                   turns into
                   (SETQ STACK (PUSH AC STACK))))


REVERSE (Definition)
      (LAMBDA (X)
       (COND ((NLISTP X) NIL)
             (T (APPEND (REVERSE (CDR X))
                        (LIST (CAR X))))))

      (Translated definition)
      (LAMBDA (X)
       (IF (LISTP X)
           (APPEND (REVERSE (CDR X))
                   (CONS (CAR X) NIL))
           NIL))


SEQUENTIAL.EXECUTION (Theorem)
      (EQUAL (EXEC (APPEND X Y) PDS ENVRN)
             (EXEC Y (EXEC X PDS ENVRN) ENVRN))
```

```
SKIPTYPE (INTERLISP Macro)
     (X (PROG1 (LIST (QUOTE IF)
                     (LIST (CADR X) (CAR X))
                     FALSE
                     (CAR PROGBODY))
               (SETQ PROGBODY (CDR PROGBODY)))
        (Comment: (SKIPTYPE x pred) instr1 instr2 ...
                  translates into
                  (IF (pred x)  FALSE instr1) instr2 ...
                  That is, the next instruction is skipped
                  if (pred x) is non-FALSE.))


STACKP (Primitive)


SUB1 (Primitive)


SUB1.0 (Axiom)
     (EQUAL (SUB1 0) 0)


SUB1.ADD1 (Axiom)
     (EQUAL (SUB1 (ADD1 X))
            (IF (NUMBERP X) X 0))


SUB1.ELIM (Axiom)
     (IMPLIES (AND (IMPLIES (NOT (NUMBERP X)) (P X 0))
                   (AND (P 0 0)
                        (IMPLIES (NUMBERP (SUB1 X))
                                 (P (ADD1 (SUB1 X)) (SUB1 X)))))
              (P X (SUB1 X)))
```

```
SUB1.LESSP (Axiom)
     (IF (NUMBERP X)
         (IF (EQUAL X 0)
             TRUE
             (LESSP (COUNT (SUB1 X)) (COUNT X)))
         TRUE)


SUB1.NNUMBERP (Axiom)
     (IF (NUMBERP X)
         TRUE
         (EQUAL (SUB1 X) 0))


TOP (Primitive)


TOP.LESSP (Axiom)
     (IF (STACKP X)
         (LESSP (COUNT (TOP X)) (COUNT X))
         TRUE)


TOP.NSTACKP (Axiom)
     (IF (NOT (STACKP X))
         (EQUAL (TOP X) 1)
         TRUE)


TOP.PUSH (Axiom)
     (EQUAL (TOP (PUSH TOP POP)) TOP)


TRUE (Primitive)


TRUE.TYPE.NO (Axiom)
     (EQUAL (TYPE.NO TRUE) 1)
```

```
UNPACK (Primitive)


UNPACK.LESSP (Axiom)
     (IF (LITATOM X)
         (LESSP (COUNT (UNPACK X)) (COUNT X))
         TRUE)


UNPACK.NLITATOM (Axiom)
     (IF (NOT (LITATOM X))
         (EQUAL (UNPACK X) NIL)
         TRUE)


UNPACK.PACK (Axiom)
     (EQUAL (UNPACK (PACK UNPACK)) UNPACK)


XCT (INTERLISP Macro)
     (X (SUBST (CAR X)
               (QUOTE INS)
               (QUOTE (SETQ AC1 (APPLY INS AC1 AC2))))
        (Comment: (XCT INS)
                  turns into
                  (SETQ AC1 (APPLY INS AC1 AC2))
                  i.e., it sets AC1 to the result of APPLYing INS to AC1
                  and AC2.))
```

8. THE PROOFS

With the exception of this page, this Section consists entirely of the theorem prover's output during the course of its proofs of the five theorems cited above. The only user interaction was in introducing the data types and function discussed above, and in commanding the theorem prover to prove the theorems in the order they are proved below.

PROOF OF THE "FORMP.OPTIMIZE" LEMMA

The conjecture to be proved is:

```
(IMPLIES (FORMP X)
         (FORMP (OPTIMIZE X)))
```

Simplification produces:

```
#1.  (IMPLIES (FORMP X)
              (FORMP (OPTIMIZE X))).
```

Give this the name #1. We'll try to prove it by induction.

There are 2 plausible inductions. These merge into one likely candidate induction. Induct on X. This induction is justified by the CAR.LESSP and CDR.LESSP inequalities.

We must now prove the following 7 goals:

```
#1.i.  (IMPLIES (AND (NOT (LISTP (CDR (CDR X))))
                     (FORMP X))
                (FORMP (OPTIMIZE X))),

#1.ii.  (IMPLIES (AND (NOT (LISTP X)) (FORMP X))
                 (FORMP (OPTIMIZE X))),

#1.iii.  (IMPLIES (AND (NOT (LISTP (CDR X))) (FORMP X))
                  (FORMP (OPTIMIZE X))),

#1.iv.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                      (LISTP X)
                      (LISTP (CDR X))
                      (NOT (FORMP (CAR (CDR (CDR X)))))
                      (NOT (FORMP (CAR (CDR X))))
                      (FORMP X))
                 (FORMP (OPTIMIZE X))),
```

```
*1.v.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                     (LISTP X)
                     (LISTP (CDR X))
                     (FORMP (OPTIMIZE (CAR (CDR (CDR X)))))
                     (NOT (FORMP (CAR (CDR X))))
                     (FORMP X))
                (FORMP (OPTIMIZE X))),

*1.vi.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                      (LISTP X)
                      (LISTP (CDR X))
                      (NOT (FORMP (CAR (CDR (CDR X)))))
                      (FORMP (OPTIMIZE (CAR (CDR X))))
                      (FORMP X))
                 (FORMP (OPTIMIZE X)))

end

*1.vii.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                       (LISTP X)
                       (LISTP (CDR X))
                       (FORMP (OPTIMIZE (CAR (CDR (CDR X)))))
                       (FORMP (OPTIMIZE (CAR (CDR X))))
                       (FORMP X))
                  (FORMP (OPTIMIZE X))).
```

Simplification produces: TRUE.

That finishes the proof of *1.

Q.E.D.

```
((SIMPLIFY 1) (PUSH 1) NEXT... (INDUCT 2 1 1 1 (X)) (SIMPLIFY 0) POP!
Q.E.D.)
Load average during proof:  2.923895
Elapsed time:  116.829 seconds
CPU time:  27.134 seconds
CONSes consumed:  10056
```

PROOF OF THE "CORRECTNESS.OF.OPTIMIZE" LEMMA

The conjecture to be proved is:

```
(IMPLIES (FORMP X)
         (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                (EVAL X ENVRN)))
```

Simplification produces:

```
*1.  (IMPLIES (FORMP X)
              (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                     (EVAL X ENVRN))).
```

Give this the name *1. We'll try to prove it by induction.

There are 3 plausible inductions. These merge into one likely candidate induction. Induct on X. This induction is justified by the CAR.LESSP and CDR.LESSP inequalities.

We must now prove the following 7 goals:

```
*1.i.  (IMPLIES (AND (NOT (LISTP (CDR (CDR X))))
                     (FORMP X))
                (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                       (EVAL X ENVRN))),

*1.ii.  (IMPLIES (AND (NOT (LISTP X)) (FORMP X))
                 (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                        (EVAL X ENVRN))),

*1.iii.  (IMPLIES (AND (NOT (LISTP (CDR X))) (FORMP X))
                  (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                         (EVAL X ENVRN))),

*1.iv.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                      (LISTP X)
                      (LISTP (CDR X))
                      (NOT (FORMP (CAR (CDR (CDR X)))))
                      (NOT (FORMP (CAR (CDR X))))
                      (FORMP X))
                 (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                        (EVAL X ENVRN))),
```

```
*1.v.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                     (LISTP X)
                     (LISTP (CDR X))
                     (EQUAL (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
                                  ENVRN)
                            (EVAL (CAR (CDR (CDR X))) ENVRN))
                     (NOT (FORMP (CAR (CDR X))))
                     (FORMP X))
                (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                       (EVAL X ENVRN))),

*1.vi.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                      (LISTP X)
                      (LISTP (CDR X))
                      (NOT (FORMP (CAR (CDR (CDR X)))))
                      (EQUAL (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
                             (EVAL (CAR (CDR X)) ENVRN))
                      (FORMP X))
                 (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                        (EVAL X ENVRN)))
```

and

```
*1.vii.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                       (LISTP X)
                       (LISTP (CDR X))
                       (EQUAL (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
                                    ENVRN)
                              (EVAL (CAR (CDR (CDR X))) ENVRN))
                       (EQUAL (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
                              (EVAL (CAR (CDR X)) ENVRN))
                       (FORMP X))
                  (EQUAL (EVAL (OPTIMIZE X) ENVRN)
                         (EVAL X ENVRN))).
```

Simplification produces the following 3 goals:

```
*1.viii.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                        (LISTP X)
                        (LISTP (CDR X))
                        (EQUAL (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
                                     ENVRN)
                               (EVAL (CAR (CDR (CDR X))) ENVRN))
                        (EQUAL (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
                               (EVAL (CAR (CDR X)) ENVRN))
                        (NOT (LISTP (CAR X)))
                        (FORMP (CAR (CDR X)))
                        (FORMP (CAR (CDR (CDR X))))
                        (NUMBERP (OPTIMIZE (CAR (CDR X))))
                        (NUMBERP (OPTIMIZE (CAR (CDR (CDR X))))))
                   (EQUAL (APPLY (CAR X)
                                 (OPTIMIZE (CAR (CDR X)))
                                 (OPTIMIZE (CAR (CDR (CDR X)))))
                          (APPLY (CAR X)
                                 (EVAL (CAR (CDR X)) ENVRN)
                                 (EVAL (CAR (CDR (CDR X))) ENVRN)))),

*1.ix.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                      (LISTP X)
                      (LISTP (CDR X))
                      (EQUAL (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
                                   ENVRN)
                             (EVAL (CAR (CDR (CDR X))) ENVRN))
                      (EQUAL (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
                             (EVAL (CAR (CDR X)) ENVRN))
                      (NOT (LISTP (CAR X)))
                      (FORMP (CAR (CDR X)))
                      (FORMP (CAR (CDR (CDR X))))
                      (NUMBERP (OPTIMIZE (CAR (CDR X))))
                      (NOT (NUMBERP (OPTIMIZE (CAR (CDR (CDR X)))))))
                 (EQUAL (APPLY (CAR X)
                               (OPTIMIZE (CAR (CDR X)))
                               (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
                                     ENVRN))
                        (APPLY (CAR X)
                               (EVAL (CAR (CDR X)) ENVRN)
                               (EVAL (CAR (CDR (CDR X))) ENVRN))))
```

and

```
*1.x.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                     (LISTP X)
                     (LISTP (CDR X))
                     (EQUAL (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
                                  ENVRN)
                            (EVAL (CAR (CDR (CDR X))) ENVRN))
                     (EQUAL (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
                            (EVAL (CAR (CDR X)) ENVRN))
                     (NOT (LISTP (CAR X)))
                     (FORMP (CAR (CDR X)))
                     (FORMP (CAR (CDR (CDR X))))
                     (NOT (NUMBERP (OPTIMIZE (CAR (CDR X))))))
                (EQUAL (APPLY (CAR X)
                              (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
                              (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
                                    ENVRN))
                       (APPLY (CAR X)
                              (EVAL (CAR (CDR X)) ENVRN)
                              (EVAL (CAR (CDR (CDR X))) ENVRN)))).
```

Cross fertilize

```
     (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
           ENVRN)
```

for

```
     (EVAL (CAR (CDR (CDR X))) ENVRN)
```

in *1.viii, and throw away the equality. This produces:

```
*1.xi.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                      (LISTP X)
                      (LISTP (CDR X))
                      (EQUAL (OPTIMIZE (CAR (CDR X)))
                             (EVAL (CAR (CDR X)) ENVRN))
                      (NOT (LISTP (CAR X)))
                      (FORMP (CAR (CDR X)))
                      (FORMP (CAR (CDR (CDR X))))
                      (NUMBERP (OPTIMIZE (CAR (CDR X))))
                      (NUMBERP (OPTIMIZE (CAR (CDR (CDR X))))))
                 (EQUAL (APPLY (CAR X)
                               (OPTIMIZE (CAR (CDR X)))
                               (OPTIMIZE (CAR (CDR (CDR X)))))
                        (APPLY (CAR X)
                               (EVAL (CAR (CDR X)) ENVRN)
                               (OPTIMIZE (CAR (CDR (CDR X))))))).
```

```
Cross fertilize
     (OPTIMIZE (CAR (CDR X)))
for
     (EVAL (CAR (CDR X)) ENVRN)
in *1.xi, and throw away the equality.  This produces:  TRUE.


Cross fertilize
@    (EVAL (CAR (CDR (CDR X))) ENVRN)
 for
     (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
           ENVRN)
in *1.ix, and throw away the equality.  This produces:

*1.xii.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                       (LISTP X)
                       (LISTP (CDR X))
                       (EQUAL (OPTIMIZE (CAR (CDR X)))
                              (EVAL (CAR (CDR X)) ENVRN))
                       (NOT (LISTP (CAR X)))
                       (FORMP (CAR (CDR X)))
                       (FORMP (CAR (CDR (CDR X))))
                       (NUMBERP (OPTIMIZE (CAR (CDR X))))
                       (NOT (NUMBERP (OPTIMIZE (CAR (CDR (CDR X)))))))
                  (EQUAL (APPLY (CAR X)
                                (OPTIMIZE (CAR (CDR X)))
                                (EVAL (CAR (CDR (CDR X))) ENVRN))
                         (APPLY (CAR X)
                                (EVAL (CAR (CDR X)) ENVRN)
                                (EVAL (CAR (CDR (CDR X))) ENVRN)))).


Cross fertilize
     (OPTIMIZE (CAR (CDR X)))
for
     (EVAL (CAR (CDR X)) ENVRN)
in *1.xii, and throw away the equality.  This produces:  TRUE.


Cross fertilize
     (EVAL (CAR (CDR (CDR X))) ENVRN)
for
     (EVAL (OPTIMIZE (CAR (CDR (CDR X))))
           ENVRN)
in *1.x, and throw away the equality.  This produces:
```

```
*1.xiii.  (IMPLIES (AND (LISTP (CDR (CDR X)))
                        (LISTP X)
                        (LISTP (CDR X))
                        (EQUAL (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
                               (EVAL (CAR (CDR X)) ENVRN))
                        (NOT (LISTP (CAR X)))
                        (FORMP (CAR (CDR X)))
                        (FORMP (CAR (CDR (CDR X))))
                        (NOT (NUMBERP (OPTIMIZE (CAR (CDR X))))))
                   (EQUAL (APPLY (CAR X)
                                 (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
                                 (EVAL (CAR (CDR (CDR X))) ENVRN))
                          (APPLY (CAR X)
                                 (EVAL (CAR (CDR X)) ENVRN)
                                 (EVAL (CAR (CDR (CDR X))) ENVRN)))).
```

Cross fertilize
    (EVAL (CAR (CDR X)) ENVRN)
for
    (EVAL (OPTIMIZE (CAR (CDR X))) ENVRN)
in *1.xiii, and throw away the equality. This produces: TRUE.

That finishes the proof of *1.

Q.E.D.

((SIMPLIFY 1) (PUSH 1) NEXT... (INDUCT 3 1 1 1 (X)) (SIMPLIFY 3) (
CROSS.FERT DELETE) (CROSS.FERT DELETE) (CROSS.FERT DELETE) (CROSS.FERT
DELETE) (CROSS.FERT DELETE) (CROSS.FERT DELETE) POP! Q.E.D.)
Load average during proof: 3.226847
Elapsed time: 339.845 seconds
CPU time: 74.993 seconds
CONSes consumed: 35933

PROOF OF THE "SEQUENTIAL.EXECUTION" LEMMA

The conjecture to be proved is:

```
(EQUAL (EXEC (APPEND X Y) PDS ENVRN)
       (EXEC Y (EXEC X PDS ENVRN) ENVRN))
```

Simplification produces:

```
*1.  (EQUAL (EXEC (APPEND X Y) PDS ENVRN)
            (EXEC Y (EXEC X PDS ENVRN) ENVRN)).
```

Give this the name *1. We'll try to prove it by induction.

There are 3 plausible inductions. These merge into 2 likely candidate inductions. However, one is more likely than the other. Induct on X (instantiating PDS). This induction is justified by the CDR.LESSP inequality.

We must now prove the following 2 goals:

```
*1.1.  (IMPLIES (NOT (LISTP X))
                (EQUAL (EXEC (APPEND X Y) PDS ENVRN)
                       (EXEC Y (EXEC X PDS ENVRN) ENVRN)))
```

and

```
*1.11.  (IMPLIES
         (AND
           (LISTP X)
           (EQUAL (EXEC (APPEND (CDR X) Y)
                        (PUSH (APPLY (CAR X)
                                     (TOP (POP PDS))
                                     (TOP PDS))
                              (POP (POP PDS)))
                        ENVRN)
                  (EXEC Y
                        (EXEC (CDR X)
                              (PUSH (APPLY (CAR X)
                                           (TOP (POP PDS))
                                           (TOP PDS))
                                    (POP (POP PDS)))
                              ENVRN)
                        ENVRN))
           (EQUAL (EXEC (APPEND (CDR X) Y)
                        (PUSH (CAR (CDR (CAR X))) PDS)
                        ENVRN)
                  (EXEC Y
                        (EXEC (CDR X)
                              (PUSH (CAR (CDR (CAR X))) PDS)
                              ENVRN)
                        ENVRN))
           (EQUAL (EXEC (APPEND (CDR X) Y)
                        (PUSH (GETVALUE (CAR (CDR (CAR X))) ENVRN)
                              PDS)
                        ENVRN)
                  (EXEC Y
                        (EXEC (CDR X)
                              (PUSH (GETVALUE (CAR (CDR (CAR X))) ENVRN)
                                    PDS)
                              ENVRN)
                        ENVRN)))
         (EQUAL (EXEC (APPEND X Y) PDS ENVRN)
                (EXEC Y (EXEC X PDS ENVRN) ENVRN))).
```

Simplification produces: TRUE.

That finishes the proof of *1.

Q.E.D.

((SIMPLIFY 1) (PUSH 1) NEXT... (INDUCT 3 2 1 1 (X / PDS)) (SIMPLIFY 0) POP! Q.E.D.)
Load average during proof: 3.023293
Elapsed time: 94.353 seconds
CPU time: 17.161 seconds
CONSes consumed: 9006

PROOF OF THE "CORRECTNESS.OF.CODEGEN" LEMMA

The conjecture to be proved is:

```
(IMPLIES (FORMP X)
         (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                      PDS ENVRN)
                (PUSH (EVAL X ENVRN)
                      (EXEC (REVERSE INS) PDS ENVRN))))
```

Simplification produces:

```
*1.  (IMPLIES (FORMP X)
              (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                           PDS ENVRN)
                     (PUSH (EVAL X ENVRN)
                           (EXEC (REVERSE INS) PDS ENVRN)))).
```

Give this the name *1. We'll try to prove it by induction.

There are 4 plausible inductions. These merge into 2 likely candidate inductions. However, one is more likely than the other. Induct on X (instantiating INS). This induction is justified by the CAR.LESSP and CDR.LESSP inequalities.

We must now prove the following 7 goals:

```
*1.i.  (IMPLIES (AND (NOT (LISTP X)) (FORMP X))
                (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                             PDS ENVRN)
                       (PUSH (EVAL X ENVRN)
                             (EXEC (REVERSE INS) PDS ENVRN)))),

*1.ii.  (IMPLIES (AND (NOT (LISTP (CDR X))) (FORMP X))
                 (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                              PDS ENVRN)
                        (PUSH (EVAL X ENVRN)
                              (EXEC (REVERSE INS) PDS ENVRN)))),
```

```
*1.iii.  (IMPLIES (AND (NOT (LISTP (CDR (CDR X))))
                       (FORMP X))
                  (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                               PDS ENVRN)
                         (PUSH (EVAL X ENVRN)
                               (EXEC (REVERSE INS) PDS ENVRN)))),

*1.iv.  (IMPLIES (AND (LISTP X)
                      (LISTP (CDR X))
                      (LISTP (CDR (CDR X)))
                      (NOT (FORMP (CAR (CDR X))))
                      (NOT (FORMP (CAR (CDR (CDR X)))))
                      (FORMP X))
                 (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                              PDS ENVRN)
                        (PUSH (EVAL X ENVRN)
                              (EXEC (REVERSE INS) PDS ENVRN)))),

*1.v.  (IMPLIES (AND (LISTP X)
                     (LISTP (CDR X))
                     (LISTP (CDR (CDR X)))
                     (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                                  PDS ENVRN)
                            (PUSH (EVAL (CAR (CDR X)) ENVRN)
                                  (EXEC (REVERSE INS) PDS ENVRN)))
                     (NOT (FORMP (CAR (CDR (CDR X)))))
                     (FORMP X))
                (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                             PDS ENVRN)
                       (PUSH (EVAL X ENVRN)
                             (EXEC (REVERSE INS) PDS ENVRN)))),

*1.vi.  (IMPLIES
         (AND
            (LISTP X)
            (LISTP (CDR X))
            (LISTP (CDR (CDR X)))
            (NOT (FORMP (CAR (CDR X))))
            (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                           (CODEGEN (CAR (CDR X)) INS)))
                         PDS ENVRN)
                   (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
                         (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                               PDS ENVRN)))
            (FORMP X))
         (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                      PDS ENVRN)
                (PUSH (EVAL X ENVRN)
                      (EXEC (REVERSE INS) PDS ENVRN))))
```

and

```
*1.vii.  (IMPLIES
          (AND
            (LISTP X)
            (LISTP (CDR X))
            (LISTP (CDR (CDR X)))
            (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                         PDS ENVRN)
                   (PUSH (EVAL (CAR (CDR X)) ENVRN)
                         (EXEC (REVERSE INS) PDS ENVRN)))
            (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                           (CODEGEN (CAR (CDR X)) INS)))
                         PDS ENVRN)
                   (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
                         (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                               PDS ENVRN)))
            (FORMP X))
          (EQUAL (EXEC (REVERSE (CODEGEN X INS))
                       PDS ENVRN)
                 (PUSH (EVAL X ENVRN)
                       (EXEC (REVERSE INS) PDS ENVRN)))).
```

Simplification produces the following 7 goals:

```
*1.viii.  (IMPLIES
            (AND (NOT (LISTP X)) (NUMBERP X))
            (EQUAL (EXEC (APPEND (REVERSE INS)
                                 (CONS (CONS (QUOTE PUSHI) (CONS X NIL))
                                       NIL))
                         PDS ENVRN)
                   (PUSH X
                         (EXEC (REVERSE INS) PDS ENVRN)))),

*1.ix.  (IMPLIES
            (AND (NOT (LISTP X))
                 (NOT (NUMBERP X)))
            (EQUAL (EXEC (APPEND (REVERSE INS)
                                 (CONS (CONS (QUOTE PUSHV) (CONS X NIL))
                                       NIL))
                         PDS ENVRN)
                   (PUSH (GETVALUE X ENVRN)
                         (EXEC (REVERSE INS) PDS ENVRN)))),
```

```
*1.x.  (IMPLIES
         (AND (NOT (LISTP (CDR X)))
              (NOT (LISTP X))
              (NUMBERP X))
         (EQUAL (EXEC (APPEND (REVERSE INS)
                              (CONS (CONS (QUOTE PUSHI) (CONS X NIL))
                                    NIL))
                      PDS ENVRN)
                (PUSH X
                      (EXEC (REVERSE INS) PDS ENVRN)))),

*1.xi.  (IMPLIES
         (AND (NOT (LISTP (CDR X)))
              (NOT (LISTP X))
              (NOT (NUMBERP X)))
         (EQUAL (EXEC (APPEND (REVERSE INS)
                              (CONS (CONS (QUOTE PUSHV) (CONS X NIL))
                                    NIL))
                      PDS ENVRN)
                (PUSH (GETVALUE X ENVRN)
                      (EXEC (REVERSE INS) PDS ENVRN)))),

*1.xii.  (IMPLIES
         (AND (NOT (LISTP (CDR (CDR X))))
              (NOT (LISTP X))
              (NUMBERP X))
         (EQUAL (EXEC (APPEND (REVERSE INS)
                              (CONS (CONS (QUOTE PUSHI) (CONS X NIL))
                                    NIL))
                      PDS ENVRN)
                (PUSH X
                      (EXEC (REVERSE INS) PDS ENVRN)))),

*1.xiii.  (IMPLIES
         (AND (NOT (LISTP (CDR (CDR X))))
              (NOT (LISTP X))
              (NOT (NUMBERP X)))
         (EQUAL (EXEC (APPEND (REVERSE INS)
                              (CONS (CONS (QUOTE PUSHV) (CONS X NIL))
                                    NIL))
                      PDS ENVRN)
                (PUSH (GETVALUE X ENVRN)
                      (EXEC (REVERSE INS) PDS ENVRN))))
```

and

```
*1.xiv.  (IMPLIES
           (AND
             (LISTP X)
             (LISTP (CDR X))
             (LISTP (CDR (CDR X)))
             (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                          PDS ENVRN)
                    (PUSH (EVAL (CAR (CDR X)) ENVRN)
                          (EXEC (REVERSE INS) PDS ENVRN)))
             (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                            (CODEGEN (CAR (CDR X)) INS)))
                          PDS ENVRN)
                    (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
                          (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                                PDS ENVRN)))
             (NOT (LISTP (CAR X)))
             (FORMP (CAR (CDR X)))
             (FORMP (CAR (CDR (CDR X)))))
           (EQUAL
            (EXEC (APPEND (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                            (CODEGEN (CAR (CDR X)) INS)))
                          (CONS (CAR X) NIL))
                  PDS ENVRN)
            (PUSH (APPLY (CAR X)
                         (EVAL (CAR (CDR X)) ENVRN)
                         (EVAL (CAR (CDR (CDR X))) ENVRN))
                  (EXEC (REVERSE INS) PDS ENVRN)))).
```

Apply the lemma SEQUENTIAL.EXECUTION to *1.viii. This produces: TRUE.

Apply the lemma SEQUENTIAL.EXECUTION to *1.ix. This produces: TRUE.

Apply the lemma SEQUENTIAL.EXECUTION to *1.x. This produces: TRUE.

Apply the lemma SEQUENTIAL.EXECUTION to *1.xi. This produces: TRUE.

Apply the lemma SEQUENTIAL.EXECUTION to *1.xii. This produces: TRUE.

Apply the lemma SEQUENTIAL.EXECUTION to *1.xiii. This produces: TRUE.

Apply the lemma SEQUENTIAL.EXECUTION to *1.xiv. This produces the following 2 goals:

```
*1.xv.  (IMPLIES
         (AND
            (LISTP X)
            (LISTP (CDR X))
            (LISTP (CDR (CDR X)))
            (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                         PDS ENVRN)
                   (PUSH (EVAL (CAR (CDR X)) ENVRN)
                         (EXEC (REVERSE INS) PDS ENVRN)))
            (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                           (CODEGEN (CAR (CDR X)) INS)))
                         PDS ENVRN)
                   (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
                         (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                               PDS ENVRN)))
            (NOT (LISTP (CAR X)))
            (FORMP (CAR (CDR X)))
            (FORMP (CAR (CDR (CDR X)))))
         (EQUAL
          (POP
              (POP (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                           (CODEGEN (CAR (CDR X)) INS)))
                         PDS ENVRN)))
          (EXEC (REVERSE INS) PDS ENVRN)))
```

and

```
*1.xvi.  (IMPLIES
           (AND
             (LISTP X)
             (LISTP (CDR X))
             (LISTP (CDR (CDR X)))
             (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                          PDS ENVRN)
                    (PUSH (EVAL (CAR (CDR X)) ENVRN)
                          (EXEC (REVERSE INS) PDS ENVRN)))
             (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                            (CODEGEN (CAR (CDR X)) INS)))
                          PDS ENVRN)
                    (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
                          (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
                                PDS ENVRN)))
             (NOT (LISTP (CAR X)))
             (FORMP (CAR (CDR X)))
             (FORMP (CAR (CDR (CDR X)))))
          (EQUAL
            (APPLY
             (CAR X)
             (TOP
               (POP (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                            (CODEGEN (CAR (CDR X)) INS)))
                          PDS ENVRN)))
             (TOP (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                          (CODEGEN (CAR (CDR X)) INS)))
                        PDS ENVRN)))
            (APPLY (CAR X)
                   (EVAL (CAR (CDR X)) ENVRN)
                   (EVAL (CAR (CDR (CDR X))) ENVRN)))).
```

Massively substitute

```
     (PUSH (EVAL (CAR (CDR X)) ENVRN)
           (EXEC (REVERSE INS) PDS ENVRN))
```

for

```
     (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
           PDS ENVRN)
```

in *1.xv, and throw away the equality. This produces:

```
*1.xvii.  (IMPLIES
           (AND
            (LISTP X)
            (LISTP (CDR X))
            (LISTP (CDR (CDR X)))
            (EQUAL (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                           (CODEGEN (CAR (CDR X)) INS)))
                         PDS ENVRN)
                   (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
                         (PUSH (EVAL (CAR (CDR X)) ENVRN)
                               (EXEC (REVERSE INS) PDS ENVRN))))
            (NOT (LISTP (CAR X)))
            (FORMP (CAR (CDR X)))
            (FORMP (CAR (CDR (CDR X)))))
           (EQUAL
            (POP
              (POP (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                           (CODEGEN (CAR (CDR X)) INS)))
                         PDS ENVRN)))
            (EXEC (REVERSE INS) PDS ENVRN))).
```

Cross fertilize

```
     (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
           (PUSH (EVAL (CAR (CDR X)) ENVRN)
                 (EXEC (REVERSE INS) PDS ENVRN)))
```

for

```
     (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                             (CODEGEN (CAR (CDR X)) INS)))
           PDS ENVRN)
```

in *1.xvii, and throw away the equality. This produces: TRUE.

Massively substitute

```
     (PUSH (EVAL (CAR (CDR X)) ENVRN)
           (EXEC (REVERSE INS) PDS ENVRN))
```

for

```
     (EXEC (REVERSE (CODEGEN (CAR (CDR X)) INS))
           PDS ENVRN)
```

in *1.xvi, and throw away the equality. This produces:

```
*1.xviii.  (IMPLIES
             (AND
              (LISTP X)
              (LISTP (CDR X))
              (LISTP (CDR (CDR X)))
              (EQUAL
                    (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                            (CODEGEN (CAR (CDR X)) INS)))
                          PDS ENVRN)
                    (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
                          (PUSH (EVAL (CAR (CDR X)) ENVRN)
                                (EXEC (REVERSE INS) PDS ENVRN))))
              (NOT (LISTP (CAR X)))
              (FORMP (CAR (CDR X)))
              (FORMP (CAR (CDR (CDR X)))))
             (EQUAL
              (APPLY
               (CAR X)
               (TOP
                (POP
                    (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                            (CODEGEN (CAR (CDR X)) INS)))
                          PDS ENVRN)))
               (TOP (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                                            (CODEGEN (CAR (CDR X)) INS)))
                          PDS ENVRN)))
              (APPLY (CAR X)
                     (EVAL (CAR (CDR X)) ENVRN)
                     (EVAL (CAR (CDR (CDR X))) ENVRN)))).
```

Cross fertilize

```
     (PUSH (EVAL (CAR (CDR (CDR X))) ENVRN)
           (PUSH (EVAL (CAR (CDR X)) ENVRN)
                 (EXEC (REVERSE INS) PDS ENVRN)))
```

for

```
     (EXEC (REVERSE (CODEGEN (CAR (CDR (CDR X)))
                             (CODEGEN (CAR (CDR X)) INS)))
           PDS ENVRN)
```

in *1.xviii, and throw away the equality. This produces: TRUE.

That finishes the proof of *1.

Q.E.D.

((SIMPLIFY 1) (PUSH 1) NEXT... (INDUCT 4 2 1 1 (X / INS)) (SIMPLIFY 7) (LEMMA SEQUENTIAL.EXECUTION) (LEMMA SEQUENTIAL.EXECUTION) (LEMMA SEQUENTIAL.EXECUTION) (LEMMA SEQUENTIAL.EXECUTION) (LEMMA SEQUENTIAL.EXECUTION) (LEMMA SEQUENTIAL.EXECUTION) (LEMMA SEQUENTIAL.EXECUTION) (MASS.SUBST DELETE) (CROSS.FERT DELETE) (MASS.SUBST DELETE) (CROSS.FERT DELETE) POP! Q.E.D.)
Load average during proof: 2.770582
Elapsed time: 330.668 seconds
CPU time: 106.12 seconds
CONSes consumed: 45178

PROOF OF THE "CORRECTNESS.OF.OPTIMIZING.COMPILER" LEMMA

The conjecture to be proved is:

```
(IMPLIES (FORMP X)
         (EQUAL (EXEC (COMPILE X) PDS ENVRN)
                (PUSH (EVAL X ENVRN) PDS)))
```

Simplification produces:

```
*1.  (IMPLIES (FORMP X)
              (EQUAL (EXEC (REVERSE (CODEGEN (OPTIMIZE X) NIL))
                           PDS ENVRN)
                     (PUSH (EVAL X ENVRN) PDS))).
```

Apply the lemmas FORMP.OPTIMIZE, CORRECTNESS.OF.CODEGEN and CORRECTNESS.OF.OPTIMIZE to *1. This produces: TRUE.

Q.E.D.

((SIMPLIFY 1) (LEMMA FORMP.OPTIMIZE CORRECTNESS.OF.CODEGEN
CORRECTNESS.OF.OPTIMIZE) Q.E.D.)
Load average during proof: 1.791236
Elapsed time: 9.757 seconds
CPU time: 4.208 seconds
CONSes consumed: 1118

9. CROSS REFERENCE TABLE FOR SECTION 7

Below we list, for each symbol defined in Section 7, all those definitions which reference it.

0	ADD1.EQUAL, ADD1.NNUMBERP, ADD1.SUB1, ADD1.TYPE.NO, CAR.NLISTP, CDR.NLISTP, CODEGEN, COMPILE, CONS.TYPE.NO, EXEC, FALSE.TYPE.NO, LESSP, OPTIMIZE, PACK.TYPE.NO, POP.NSTACKP, PUSH.TYPE.NO, REVERSE, SUB1.0, SUB1.ADD1, SUB1.ELIM, SUB1.LESSP, SUB1.NNUMBERP, TOP.NSTACKP, TRUE.TYPE.NO, UNPACK.NLITATOM.
AC	HLRZ, HRRZ, LOAD@, POPARG, PUSHARG.
AC1	EXEC, XCT.
AC2	EXEC, XCT.
ADD1	ADD1.EQUAL, ADD1.NNUMBERP, ADD1.SUB1, ADD1.TYPE.NO, CODEGEN, CONS.TYPE.NO, EXEC, PACK.TYPE.NO, PUSH.TYPE.NO, SUB1.ADD1, SUB1.ELIM, TOP.NSTACKP, TRUE.TYPE.NO.
AND	ADD1.TYPE.NO, CONS.EQUAL, CONS.TYPE.NO, FORMP, OPTIMIZE, PACK.TYPE.NO, PUSH.EQUAL, PUSH.TYPE.NO, SUB1.ELIM.
APPEND	APPEND, REVERSE, SEQUENTIAL.EXECUTION.
APPLY	EVAL, EXEC, NUMBERP.APPLY, OPTIMIZE, XCT.
ASSEMBLE	EXEC.
CADDR	CODEGEN, EVAL, FORMP, OPTIMIZE.
CADR	CAIE, CODEGEN, EVAL, FORMP, HRRZ, OPTIMIZE, SKIPTYPE.
CAIE	EXEC.
CAR	APPEND, CAIE, CAR.CONS, CAR.LESSP, CAR.NLISTP, CDR.CONS, CODEGEN, CONS.EQUAL, CONS.TYPE.NO, EVAL, EXEC, FORMP, HLRZ, ILDI, OPTIMIZE, REVERSE, SKIPTYPE, XCT.
CAR'	CONS.EQUAL.

CDDR	FORMP.
CDR	APPEND, CAIE, CAR.CONS, CDR.CONS, CDR.LESSP, CDR.NLISTP, CODEGEN, CONS.EQUAL, CONS.TYPE.NO, EVAL, EXEC, FORMP, ILDI, OPTIMIZE, REVERSE, SKIPTYPE.
CDR'	CONS.EQUAL.
CODEGEN	CODEGEN, COMPILE, CORRECTNESS.OF.CODEGEN.
COMPILE	CORRECTNESS.OF.OPTIMIZING.COMPILER.
COND	APPEND, CODEGEN, EVAL, FORMP, OPTIMIZE, REVERSE.
CONS	APPEND, ASSEMBLE, CAR.CONS, CDR.CONS, CODEGEN, CONS.EQUAL, CONS.TYPE.NO, JUMP, OPTIMIZE, REVERSE.
COUNT	CAR.LESSP, CDR.LESSP, POP.LESSP, SUB1.LESSP, TOP.LESSP, UNPACK.LESSP.
DECODE	EXEC.
ENVRN	CORRECTNESS.OF.CODEGEN, CORRECTNESS.OF.OPTIMIZE, CORRECTNESS.OF.OPTIMIZING.COMPILER, EVAL, EXEC, LOAD@, SEQUENTIAL.EXECUTION.
EQUAL	ADD1.EQUAL, ADD1.NNUMBERP, ADD1.SUB1, ADD1.TYPE.NO, CAIE, CAR.CONS, CAR.NLISTP, CDR.CONS, CDR.NLISTP, CONS.EQUAL, CONS.TYPE.NO, CORRECTNESS.OF.CODEGEN, CORRECTNESS.OF.OPTIMIZE, CORRECTNESS.OF.OPTIMIZING.COMPILER, EXEC, FALSE.TYPE.NO, LESSP, PACK.EQUAL, PACK.TYPE.NO, POP.NSTACKP, POP.PUSH, PUSH.EQUAL, PUSH.TYPE.NO, SEQUENTIAL.EXECUTION, SUB1.0, SUB1.ADD1, SUB1.LESSP, SUB1.NNUMBERP, TOP.NSTACKP, TOP.PUSH, TRUE.TYPE.NO, UNPACK.NLITATOM, UNPACK.PACK.
EVAL	CORRECTNESS.OF.CODEGEN, CORRECTNESS.OF.OPTIMIZE, CORRECTNESS.OF.OPTIMIZING.COMPILER, EVAL.
EXEC	CORRECTNESS.OF.CODEGEN, CORRECTNESS.OF.OPTIMIZING.COMPILER, EXEC, SEQUENTIAL.EXECUTION.
FALSE	AND, CAIE, FALSE.TYPE.NO, FORMP, IMPLIES, LESSP, NLISTP, NOT, OR, SKIPTYPE.
FN	NUMBERP.APPLY.

FORM	CODEGEN, COMPILE, EVAL, OPTIMIZE.
FORMP	CORRECTNESS.OF.CODEGEN, CORRECTNESS.OF.OPTIMIZE, CORRECTNESS.OF.OPTIMIZING.COMPILER, FORMP, FORMP.OPTIMIZE.
GETVALUE	EVAL, EXEC, LOAD@.
GO	JUMP.
HLRZ	EXEC.
HRRZ	EXEC.
IF	ADD1.EQUAL, ADD1.NNUMBERP, ADD1.SUB1, AND, APPEND, CAIE, CAR.LESSP, CAR.NLISTP, CDR.LESSP, CDR.NLISTP, CODEGEN, EVAL, EXEC, FORMP, IMPLIES, LESSP, NLISTP, NOT, OPTIMIZE, OR, POP.LESSP, POP.NSTACKP, REVERSE, SKIPTYPE, SUB1.ADD1, SUB1.LESSP, SUB1.NNUMBERP, TOP.LESSP, TOP.NSTACKP, UNPACK.LESSP, UNPACK.NLITATOM.
ILDI	EXEC.
IMPLIES	CORRECTNESS.OF.CODEGEN, CORRECTNESS.OF.OPTIMIZE, CORRECTNESS.OF.OPTIMIZING.COMPILER, FORMP.OPTIMIZE, SUB1.ELIM.
INS	CODEGEN, CORRECTNESS.OF.CODEGEN, ILDI, XCT.
INSTR	EXEC.
JUMP	EXEC.
KWOTE	CAIE.
LAMBDA	AND, APPEND, CODEGEN, COMPILE, EVAL, EXEC, FORMP, IMPLIES, LESSP, NLISTP, NOT, OPTIMIZE, OR, REVERSE.
LESSP	CAR.LESSP, CDR.LESSP, LESSP, POP.LESSP, SUB1.LESSP, TOP.LESSP, UNPACK.LESSP.
LIST	CAIE, CODEGEN, OPTIMIZE, REVERSE, SKIPTYPE.
LISTP	APPEND, CAR.LESSP, CAR.NLISTP, CDR.LESSP, CDR.NLISTP, CODEGEN, CONS.TYPE.NO, EVAL, EXEC, FORMP, NLISTP, OPTIMIZE, REVERSE.
LITATOM	PACK.TYPE.NO, UNPACK.LESSP, UNPACK.NLITATOM.

PUSHARG	EXEC.
PUSHI	CODEGEN, EXEC.
PUSHV	CODEGEN.
Q	AND, IMPLIES, OR.
QUOTE	ASSEMBLE, CAIE, CODEGEN, HLRZ, HRRZ, ILDI, JUMP, LOAD@, POPARG, PUSHARG, SKIPTYPE, XCT.
RETURN	EXEC, OPTIMIZE.
REVERSE	COMPILE, CORRECTNESS.OF.CODEGEN, REVERSE.
SETQ	CAIE, HLRZ, HRRZ, ILDI, LOAD@, OPTIMIZE, POPARG, PUSHARG, SKIPTYPE, XCT.
SKIPTYPE	EXEC.
STACK	POPARG, PUSHARG.
STACKP	POP.LESSP, POP.NSTACKP, PUSH.TYPE.NO, TOP.LESSP, TOP.NSTACKP.
SUB1	ADD1.SUB1, ADD1.TYPE.NO, LESSP, SUB1.0, SUB1.ADD1, SUB1.ELIM, SUB1.LESSP, SUB1.NNUMBERP.
SUBPAIR	HLRZ, HRRZ, ILDI, LOAD@, POPARG, PUSHARG.
SUBST	XCT.
T	APPEND, CODEGEN, EVAL, FORMP, OPTIMIZE, REVERSE.
TEMP1	OPTIMIZE.
TEMP2	OPTIMIZE.
TOP	EXEC, POP.PUSH, POPARG, PUSH.EQUAL, PUSH.TYPE.NO, TOP.LESSP, TOP.NSTACKP, TOP.PUSH.
TOP'	PUSH.EQUAL.
TRUE	ADD1.EQUAL, ADD1.NNUMBERP, AND, CAR.LESSP, CAR.NLISTP, CDR.LESSP, CDR.NLISTP, FORMP, IMPLIES, LESSP, NLISTP, NOT, OR, POP.LESSP, POP.NSTACKP, SUB1.LESSP, SUB1.NNUMBERP, TOP.LESSP, TOP.NSTACKP, TRUE.TYPE.NO, UNPACK.LESSP, UNPACK.NLITATOM.

TYPE.NO	ADD1.TYPE.NO, CONS.TYPE.NO, FALSE.TYPE.NO, PACK.TYPE.NO, PUSH.TYPE.NO, TRUE.TYPE.NO.
UNPACK	PACK.EQUAL, PACK.TYPE.NO, UNPACK.LESSP, UNPACK.NLITATOM, UNPACK.PACK.
UNPACK'	PACK.EQUAL.
VAR	LOAD@.
X	ADD1.EQUAL, ADD1.NNUMBERP, ADD1.SUB1, ADD1.TYPE.NO, APPEND, ASSEMBLE, CAIE, CAR.LESSP, CAR.NLISTP, CDR.LESSP, CDR.NLISTP, CONS.TYPE.NO, CORRECTNESS.OF.CODEGEN, CORRECTNESS.OF.OPTIMIZE, CORRECTNESS.OF.OPTIMIZING.COMPILER, FORMP, FORMP.OPTIMIZE, HLRZ, HRRZ, ILDI, JUMP, LESSP, LOAD@, NLISTP, NUMBERP.APPLY, PACK.TYPE.NO, POP.LESSP, POP.NSTACKP, POPARG, PUSH.TYPE.NO, PUSHARG, REVERSE, SEQUENTIAL.EXECUTION, SKIPTYPE, SUB1.ADD1, SUB1.ELIM, SUB1.LESSP, SUB1.NNUMBERP, TOP.LESSP, TOP.NSTTACKP, UNPACK.LESSP, UNPACK.NLITATOM, XCT.
XCT	EXEC.
Y	ADD1.EQUAL, APPEND, LESSP, NUMBERP.APPLY, SEQUENTIAL.EXECUTION.

REFERENCES

1. Boyer, R. S., and Moore, J S. Proving theorems about LISP functions, Journal of the Associating for Computing Machinery, Vol. 22, No. 1, January, 1975, ppp. 129-144.

2. Burstall, R. M. Proving properties of programs by structural induction, The Computer Journal, Vol. 12, No. 1, February 1969, pp. 41-48.

3. Cartwright, R. User-defined data types as an aid to verifying LISP programs, Informal technical memo, Artificial Intelligence Laboratory, Stanford University, 1976.

4. Diffie, W. (unpublished disk file) Artificial Intelligence Laboratory, Stanford University, 1972.

5. McCarthy, J. and Painter, J. Correctness of a compiler for arithmetic expressions, Mathematical Aspects of Computer Science, Vol. XIX, Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Providence, Rhode Island, 1967, pp. 33-41.

6. Milner, R. and Weyhrauch, R. Proving compiler correctness in a mechanized logic, Machine Intelligence 7 (eds. B. Meltzer and D. Michie) Edinburgh University Press, 1972, pp. 51-70

DISTRIBUTION LIST

The below listing is the official distribution list for the technical, annual, and final reports for Contract N00014-75-C-0816.

Defense Documentation Center Cameron Station Alexandria, VA. 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, VA. 22217	2 copies
Office of Naval Research Code 102IP Arlington, VA. 22217	6 copies
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, MASS. 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, ILL. 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, CA. 91106	1 copy
New York Area Office 715 Broadway - 5th Floor New York, N.Y. 10003	1 copy
Assistant Chief for Technology Office of Naval Research Code 200 Arlington, VA. 22217	1 copy
Naval Research Laboratory Technical Information Div., Code 2627 Washington, D.C. 20375	6 copies
Dr. A. L. Slafkosky Scientific Advisor Commandant of the Marine Corps (Code RD-1) Washington, D.C. 20380	1 copy
Office of Naval Research Code 455 Arlington, VA. 22217	1 copy
Office of Naval Research Code 458 Arlington, VA. 22217	1 copy
Naval Elec. Laboratory Center Advanced Software Tech. Div. Code 5200 San Diego, CA. 92152	1 copy
Mr. E. H. Gleissner Naval Ship Res. & Dev. Center Computation & Math. Dept. Bethesda, MD. 20084	1 copy
Captain Grace M. Hopper NAICOM/MIS Planning Branch (OP-916D) Office of Chief of Naval Operations Washington, D.C. 20350	1 copy
Mr. Kin B. Thompson Technical Director Information Sys. Div. (OP-91T) Office of Chief of Naval Operations Washington, D.C. 20350	1 copy
Officer-in-Charge Naval Surface Weapons Center Dahlgren Laboratory Dahlgren, VA. 22448 Attn: Code KP	1 copy